An experimental validation of the PRO model for parallel and distributed computation

Mohamed Essaïdi, Jens Gustedt

An experimental validation of the PRO model for parallel and distributed computation

Mohamed Essaïdi
INRIA Sophia-Antipolis, France
email: Mohamed.Essaidi@sophia.inria.fr

Jens Gustedt
INRIA Lorraine & LORIA, France
email: Jens.Gustedt@loria.fr

***Abstract*— The Parallel Resource-Optimal (PRO) computation model was introduced by Gebremedhin et al. [2002] as a framework for the design and analysis of efficient parallel algorithms. The key features of the PRO model that distinguish it from previous parallel computation models are the full integration of *resource-optimality* into the design process and the use of a *granularity function* as a parameter for measuring quality. In this paper we present experimental results on parallel algorithms, designed using the PRO model, for two representative problems: list ranking and sorting. The algorithms are implemented using *SSCRAP*, our environment for developing coarse-grained algorithms. The experimental performance results observed agree well with analytical predictions using the PRO model. Moreover, by using different platforms to run our experiments, we have been able to provide an integrated view of the modeling of an underlying architecture and the design and implementation of scalable parallel algorithms.**

I. Introduction

A faithful model of a target machine architecture forms the basis for the design of efficient algorithms and portable programs. Classical models for parallel or distributed computing (such as the PRAM or network of processors) have been found to be of limited use for such goals. These models are often either too generic – failing to provide valid predictions – or too specific – failing to allow easy portability.

This issue was addressed by Valiant [1990] when he introduced the BSP model, a model for more realistic architectures consisting of processors each of which possesses a private memory of substantial size. In the following we will refer to such architectures as *coarse grained*, a notion that covers a large portion of currently available computing environments, including parallel mainframes and clusters.

The Parallel Resource-Optimal (PRO) model, introduced in Gebremedhin et al. [2002], is similar to the BSP model in terms of its assumption on machine architecture and execution model. In particular, it assumes a network of processors and a sequence of alternating computation and communication *supersteps*. Unlike the BSP model, however, it restricts attention to a set of algorithms that are likely to be efficient in practice. Besides, it uses fewer and simpler parameters for algorithm analysis.

The aim of this paper is to provide experimental evidences that help validate the relevance of the PRO model. Whereas the complexity of a particular algorithm can be analytically shown using a specific model, a validation of the model itself requires experiments. To be convincing, an experimental validation needs to fulfill several criteria:

- it must cover a sufficiently large selection of algorithms, inputs, and platforms;
- it must address the practical aspects of the model;
- and it should be reproducible.

Moreover, a parallel implementation needs to be compared with a good quality *sequential* implementation to justify its usefulness. This later criterion is not easy to fulfill, and many proposals in the literature that are appealing at a first sight miss the comparison to the sequential setting by several orders of magnitude, see e.g Chan and Dehne [2003].

This work attempts to fulfill all of the aforementioned criteria. Before presenting the experimental settings and results, we briefly discuss the PRO model in the rest of the current section. For a more detailed discussion of the PRO model, see Gebremedhin et al. [2002].

The PRO model

The key features of PRO that distinguish if from other models for coarse grained architectures are *relativity* and *resource optimality* and a new quality measure in terms of *granularity*. Relativity refers to the fact that the design and analysis of a parallel algorithm is done relative to the time and space complexity of a specific sequential algorithm. A PRO algorithm is required to be both time and space optimal (hence resource optimal). A parallel algorithms is said to be optimal w.r.t the use of some specific resource if the overall cost of the algorithm for that resource is proportional to the one of the reference sequential algorithm. Hence, the optimality requirement here restricts attention to algorithms for which the cost of parallelization remains within reasonable bounds. By the same token, it excludes existing algorithms that entail non-linear costs for parallelization: the potential use of such algorithms is restricted to cases where other criteria than performance are set.

The PRO model is defined as a framework for the design and analysis of practical, optimal and scalable parallel algorithm relative to a specific sequential algorithms. Let Time(n) and Space(n) denote the time and space complexity of the considered sequential algorithm for a given problem with input size n and let $Grain(n)$ be a function of n. The PRO model is defined to have the following components.

Machine Model: The underlying machine is assumed to consist of p processors that are interconnected by a router, network or communication bus that can deliver point-to-point messages. The size of any individual message is not fixed but may consist of several machine words. Each processor possesses a private memory of size $M = O(\frac{Space(n)}{p})$ and it is this value M that constitutes the only restriction for the size of an individual message. This requirement enforces *space optimality* of an algorithm.

Execution Model: For any value $p = O(Grain(n))$ a PRO algorithm, consist of $o(\frac{Time(n)}{p^2})$ supersteps. In each superstep, each processor

- performs computations on data stored in private memory,
- sends at most one message to every other processor,
- sends and receives at most M words in total,
- processors are not required to be explicitly synchronized (by barrier mechanism) at the end of each superstep.

The parallel runtime is required to be in $O(\frac{Time(n)}{p})$. This ensures *work* and *time optimality*.

Quality Measure: Since the PRO model has a built-in resource optimality requirement, the use of resources will not distinguish different PRO algorithms. Instead, the quality of a PRO algorithm $\mathcal{A}$ is measured using a *granularity* function $Grain_{\mathcal{A}}(n)$. This is a function in the input size n and its value gives the maximum number of processors p that can be employed while still satisfying the PRO-requirements on resource utilization. By that measure, the more processors that can be used efficiently for a given input size, the more an algorithm is considered to be scalable.

The granularity function of a PRO algorithm cannot be seen in isolation from the number of supersteps in the algorithm. In fact, it has been shown by Gebremedhin et al. [2002] that the maximum number of supersteps that an algorithm should perform is $O(\sqrt{n})$. If this restriction is observed the communication time of an application will only be sensible to the bandwidth of the interconnection network. Its waiting time for network latency, message startup *etc.* will be orders of magnitude smaller. This restriction on the supersteps then implies that the maximal granularity of a PRO algorithm in turn may be at most $O(\sqrt{n})$, too.

Overview

This paper is organized as follows. We first describe the considered parallel algorithms in Section II. We evaluate their theoretical computation and communication complexity to prove that they fully comply with all PRO requirements. Section II-C gives an overview over the considered experimental environment by presenting the implementation tools, platforms and test settings. In Section III we then analyze the obtained results. We mainly focus on the efficiency, realism, predictability and memory usage of PRO algorithms. In particular, in Section III-C we discuss the observable dependency between the algorithmic efficiency and the granularity as it is considered as the PRO quality measure. Finally, we conclude in Section IV by giving an outlook to current and future work.

II. Experimental studies of PRO

When aiming to validate a modeling by experiments the algorithms that are benchmarked must be quite carefully chosen. Here we present two algorithms which we think are good representatives for two different classes. The first, *list ranking*, is an algorithm on a highly irregular data structure. We will see that here computation and communication are of the same order of magnitude. The second, sorting, uses some regularly structured data, namely tables, but has a communication cost that is slightly less than the computation cost.

To be able to demonstrate the conformity of both algorithms to the PRO model, we evaluate their computational cost, communication volume, number of supersteps and theoretical absolute speedup. Then, we also detail the experimental environment by introducing the implementation library and by giving a description of the used platforms.

A. List Ranking

The *list ranking* problem appears frequently, see e.g Caceres et al. [1997], in parallel algorithms that compute on object like lists, trees or graphs. List Ranking has a linked list of elements as input where each element knows its successor as well as the distance which separates these two elements. Solving the List Ranking problems consists in computing for each element the distance which separates it from the last node in the list. In contrast to the known theoretical complexity of this problem, is notoriously difficult to implement solutions with acceptable speedups on few processors, see e.g. Sibeyn [1999].

In this paper, we consider the randomized List Ranking algorithm proposed in Guérin Lassous and Gustedt [2002]. This algorithm is based on the recursive construction of independent sets. An independent set I of the elements list L is a subset of L for such that no two elements in I are neighbors in L. If we consider that in parallel approach the list elements are randomly distributed over processors, the randomized List Ranking algorithm using the independent set technique corresponds to Algorithm 1.

Algorithm 1 is randomized since the selection of the independent set I is based on a randomized election of the elements in the considered independent set. In this paper we will not detail the selection mechanism of the independent set which is fully described in Guérin Lassous and Gustedt [2002].

If we suppose that there is $0 < \varepsilon < 1$ verifying on each recursion $|I| \geq \varepsilon|L|$, then the recursion depth (number of supersteps) of Algorithm 1 is in $O(\log_{1/(1-\varepsilon)}(|L|))$. This follows from the convergence of the geometric series $\sum_i \varepsilon^i$ for any $0 < \varepsilon < 1$. In addition, if we set $small = n/p$, the recursion depth of the algorithm becomes then $O(\log_{1/(1-\varepsilon)}(p))$: in each recursion i the total size n_i of the list L is reduced to at least $(1-\varepsilon)n_i$ elements, $O(\log_{1/(1-\varepsilon)}(p))$ recursions are required to reach n/p elements (sequentially ranked in the last recursion).

Algorithm 1: List Ranking by Independent set

Input: Doubly linked list L and randomly distributed among the processors. Each element v knows its right neighbor $r(v)$, left one $l(v)$ and the distance $distr(v)$ to its right neighbor $r(v)$

Output: For each element v the distance $d(v)$ to the end of the list

begin

  if $|L| \leq small$ **then**

    send L to Processor 0;

    sequential List Ranking on Processor 0;

  else

    Let I be an independent set in L with only internal elements and $D = L \setminus I$;

    foreach $v \in I$ **do**

      Send $(l(v))$ to $r(v)$;

      Send $(r(v)$ and $distr(v))$ to $l(v)$;

    foreach $v \in D$ **do**

      if $l(v) \in I$ **then**

        Save $l(v)$ in $oldl(v)$ and set $l(v)$ to the new value received from $old(v)$;

      if $r(v) \in I$ **then**

        Let nr and nd be the values received from $r(v)$;

        Set $r(v) = nr$ and $distr(v) += nd$;

Recursion IndRanking(D);

    foreach $v \in D$ *with* $oldl(v) \in I$ **do**

      Send $distr(v)$ to $oldl(v)$;

    foreach $v \in I$ **do**

      Let nd be the value Received from $r(v)$;

      Set $d(v) = distr(v) + nd$;

end

Note that every list element is a member of the independent set at most once. In such case it requires a constant number of communications (at most two communications). Then the overall communication volume required by the algorithm is $O(n)$. As for the communication, each element of the independent set requires a constant number of instructions to be executed. Thus, the overall theoretical computational complexity of the algorithm will be in $O(n)$. If the construction of the independent set ensures a balanced distribution among the processors, the computational cost of the parallel algorithm will be in $O(n/p)$ and then the obtained speedup will be in $O(p)$.

Since the number of supersteps is of the same magnitude as the recursion depth, it is $O(\log n)$ and thus the algorithms respects all restrictions of the PRO model.

B. Sorting

Like List Ranking, sorting problems occur frequently in sequential an distributed computing. For our experimental studies, we chose the randomized and distributed sorting algorithm described by Gerbessiotis and Valiant [1994]. This algorithm, initially described in the BSP model and based on an over-sampling technique, corresponds to Algorithm 2.

Algorithm 2: Distributed Sorting

Input: $0 \leq \rho < p$ a number identifying this processor, T a distributed array of values, T_ρ corresponds to the local sub-array for the current processor

Result: The T array is globally sorted.

begin

Φ_1 Randomly extract a sample E_ρ of k values from T_ρ;

  Send the sample E_ρ to Processor 0;

  if $\rho = 0$ **then**

    $E \leftarrow \bigcup_{0 \leq i < p} E_i$;

Φ_2 local_sort(E);

    Create an array of splitters S and set $S[0] = -\infty$;

    foreach $i = 1, \ldots, p-1$ **do** $S[i] \leftarrow E[i \times k]$;

    Broadcast S to all the other processors;

  Receive the splitters S from Processor 0;

Φ_3 **foreach** *Value* $v \in T_\rho$ **do**

    find ℓ with $S[\ell - 1] \leq v < S[\ell]$;

    $M_\ell \leftarrow M_\ell \cup \{v\}$;

  foreach $i = 0, \ldots, p-1$ **do**

    Send M_i to Processor i;

  foreach $i = 0, \ldots, p-1$ **do**

    Receive array M'_i from processor i;

Φ_4 $T_\rho \leftarrow \bigcup_{0 \leq i < p} M'_i$;

  local_sort(T_ρ);

end

In Algorithm 2, we can easily distinguish 4 phases:

- Φ_1**:** randomized and parallel sampling. This can be done in $O(n/p)$.
- Φ_2**:** sequential sorting on Processor 0 of all received samples. If $p \cdot k \leq n/p$, the computation cost of the sequential sorting is at most in $O(m \log(m))$ with $m = n/p$.
- Φ_3**:** parallel ranking of all local values according to p splitters. By using binary search, this phase can be done in $O(m \log(p))$ time.
- Φ_4**:** local sorting (in parallel). If we assume that the three first phases provide a balanced redistribution of the initial array (n/p values per processor), the computation cost of the last sorting phase is in $O(m \log(m))$.

So the overall computation cost of Algorithm 2 is:

$$O(m(\log(m) + \log(p))) = O(m \log(n)) = O(\frac{n}{p} \log(n)).$$

Since the complexity of the best sequential sorting algorithms is in $\Omega(n \log(n))$, the theoretical absolute speed-up of the given algorithm is in $O(p)$. The communication at the end of the third phase is by far the most expensive: the initial global array T (size n) is completely redistributed through the interconnection network. The overall communication volume required by the algorithm can be expressed by $O(n)$.

Since the number of supersteps of the algorithm is also

Platform name	Type	Proc Nb.	Proc. MHz	Memory (GiB)	Network type	Bandwidth Mb/s	OS
SGI Origin3000	DSM ccNUMA	56	700	42	SGI NUMA-Link		IRIX
SunFire 6800	DSM ccNUMA	24	900	24	Sun Fireplane Interconnect		Solaris
Icluster	Cluster	200	733	51.2	Ethernet	100	Linux
Albus	SMP Cluster	16	1333	8	Ethernet Myrinet	100 4000	Linux

TABLE I
CONSIDERED PLATFORMS

suitably bounded (by a constant in that case) this algorithm fulfills the PRO-requirements, too.

C. Experimental setting

Both algorithms, List Ranking and Sorting, described above are implemented by using a development environment for coarse grained algorithms called *SSCRAP* Essaïdi et al. [2002, 2004]. *SSCRAP* (Soft Synchronized Computing in Rounds for Adequate Parallelization) is a C++ communication and synchronization library for the implementation of parallel algorithms on coarse grained architectures, in particular clusters *and* parallel machines. *SSCRAP* supports all known so-called coarse grained models and in particular PRO model. Indeed, it allows the efficient implementation of algorithms which are designed for the different flavors of these models by supporting, at the same time, their respective execution models. Providing a high level of abstraction, *SSCRAP* handles demanding communications transparent for the user and handles data exchanges and inter-process synchronization efficiently.

Thanks to its efficiency, its low overhead and its architectural independence, *SSCRAP* can be used to carry out accurate experimental studies for several coarse grained algorithms and for the coarse grained models themselves.

For our experiments we consider four platforms for which we summarize the main characteristics in Table I. In this table, we indicate the platform name used in this paper, the architecture type, the number of available processors, the processor frequency, the total memory size, the interconnection type, the communication bandwidth and the operating system. We used two different types of platforms: DSM machines and clusters. For DSM, we there are two different 64 bit machines. The first one is an SGI Origin 3000 and the second is a SunFire 6800 machine. In addition, we had the opportunity to experiment the SGI machine with two different sets of processors, the first of type R 12000 and the second of type R 16000, to which we refer as R12M and R16M respectively.

Table I also presents two clusters. The first one, "Icluster" was a large PC cluster with about 200 common desktops powered by PIII processor. The second cluster, "Albus", is a cluster composed of 8 biprocessor-AMD Athlon MP SMP nodes. "Albus" has two different interconnections, a standard 100 Mb/s switched ethernet and a high speed Myrinet.

III. EXPERIMENTAL ANALYSIS

Our experiments have been carried out for a large scale of values on the different platforms, a more complete impression of the overall setting is given by Essaïdi [2004]. Here we will concentrate on some typical curves that emphasize on the aspects that validate the computational model.

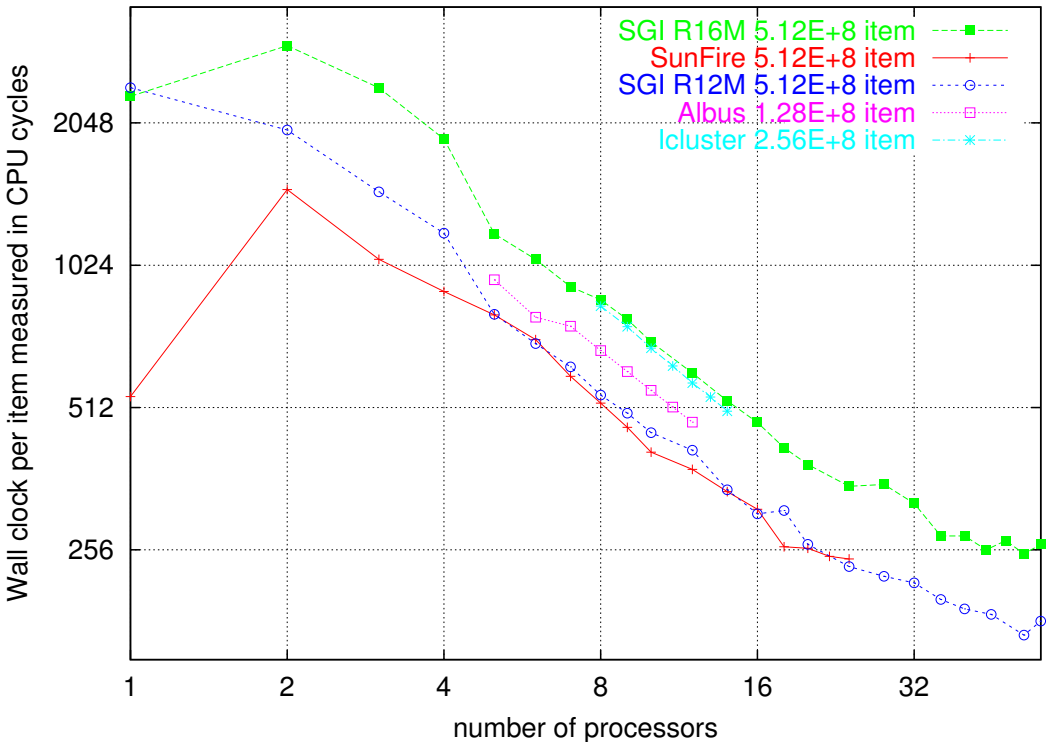
Figure 1 shows some typical curves for the execution times of the chosen algorithms. On a doubly logarithmic scale, they plot the *number of items* against the *execution times* per item. The term "number of items" qualifies the number of the list elements for the List Ranking and the number of elements to be sorted for the Sorting algorithms. To have a comparable behavior between the different architectures, execution times are normalized with the CPU frequency such that they appear as clock cycles of the underlying architecture. In a certain sense this normalization even hides some apparent differences between the platforms in terms of efficiency when comparing pure running times. But it should be easy to deduce these times from the clock frequencies as given in Table I.

For each four-tuple (algorithm, platform, number of items, number of processors), the given results correspond to the average of 10 runs. We note here, that in all cases, the variance is very low. In addition, we note that the executions on one processor correspond to those of the optimal sequential algorithm and not to those of the parallel algorithm executed on a single processor. Therefore, all the obtained speedups are not relative but absolute.

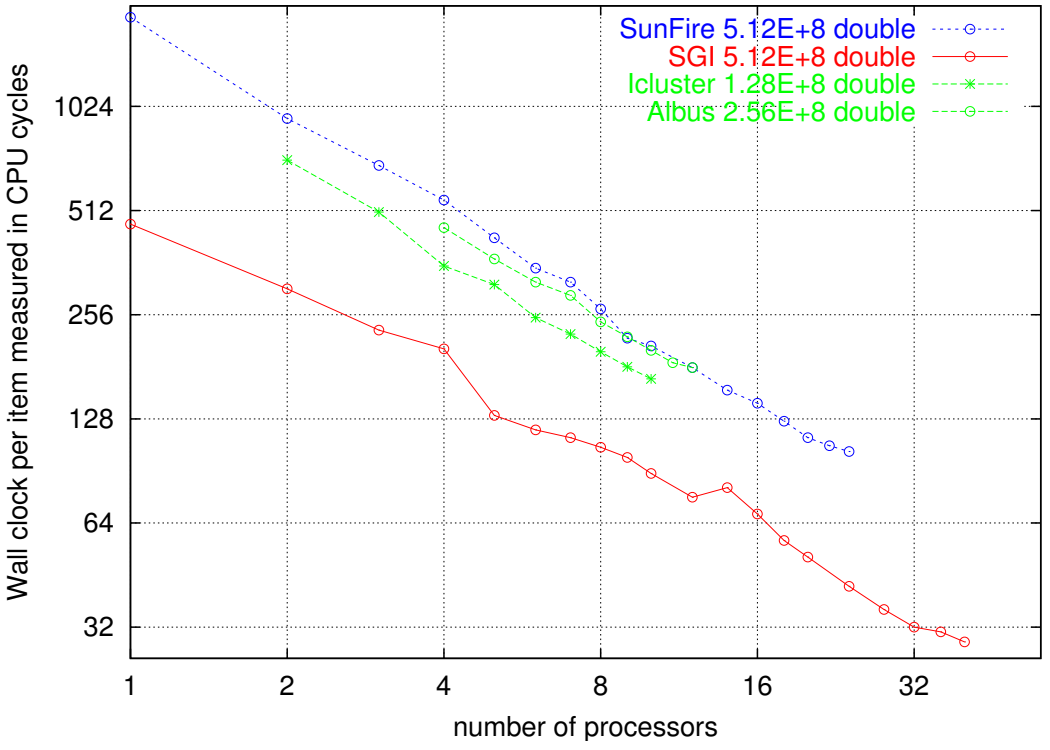
A. Efficiency, realism and predictability

Figure 1 shows the combined results of respectively list ranking and sorting algorithms. Each figure gathers the results of all considered platforms for the largest computed input size. For both algorithms, we can clearly notice that:

1) Since the shown curves are close to straight lines, the speedup is linear in a wide range of processors.
2) By comparing the optimal sequential algorithm execution to the parallel execution using two processors, it is easy to deduce that the overhead for parallelization for most architectures (except SunFire) is relatively small.
3) The curves for execution time are almost parallel to each other.
4) For both algorithms, the behavior is noticeably similar on the various platforms.



(a) List Ranking



(b) Sorting

Fig. 1. Results of all platforms combined. Ideal speedup would correspond to curves of slope -1.

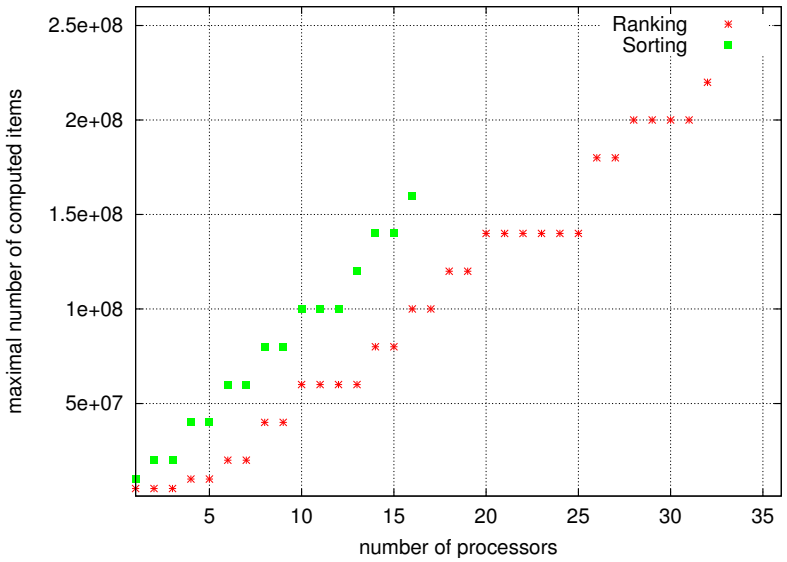


Fig. 2. List Ranking and Sorting: Maximal entry size computed on Icluster

The two first observations clearly indicate that the experiments for both algorithms confirm the PRO speedup criterion and show the realism of PRO approach. With the other observations, we see that the implementation is able to reproduce the behavior that is claimed by the analysis within the PRO model. Thus PRO modeling and algorithmic design provides an architectural independent framework. Consider a given PRO algorithm and fully detailed description on an architecture "B". Then, with some initial results (execution times for example) on an other architecture "A" PRO makes a realistic prediction of the behavior of the same algorithm for "A" possible. However, the highlighted realism and predictability provided by the PRO modeling and design strongly depend on the quality and the overhead of the considered implementation.

B. PRO memory scalability and usage

Let N_{seq} the maximal input size for one of the algorithms that can be computed sequentially in memory of size M_{seq}. Following the above detailed analysis we see that Sorting and List Ranking executions on p nodes can process $\Omega(p \cdot N_{seq})$ input size in a memory volume of $\Theta(p \cdot M_{seq})$. So we see that these algorithms are also resource optimal in the use of memory.

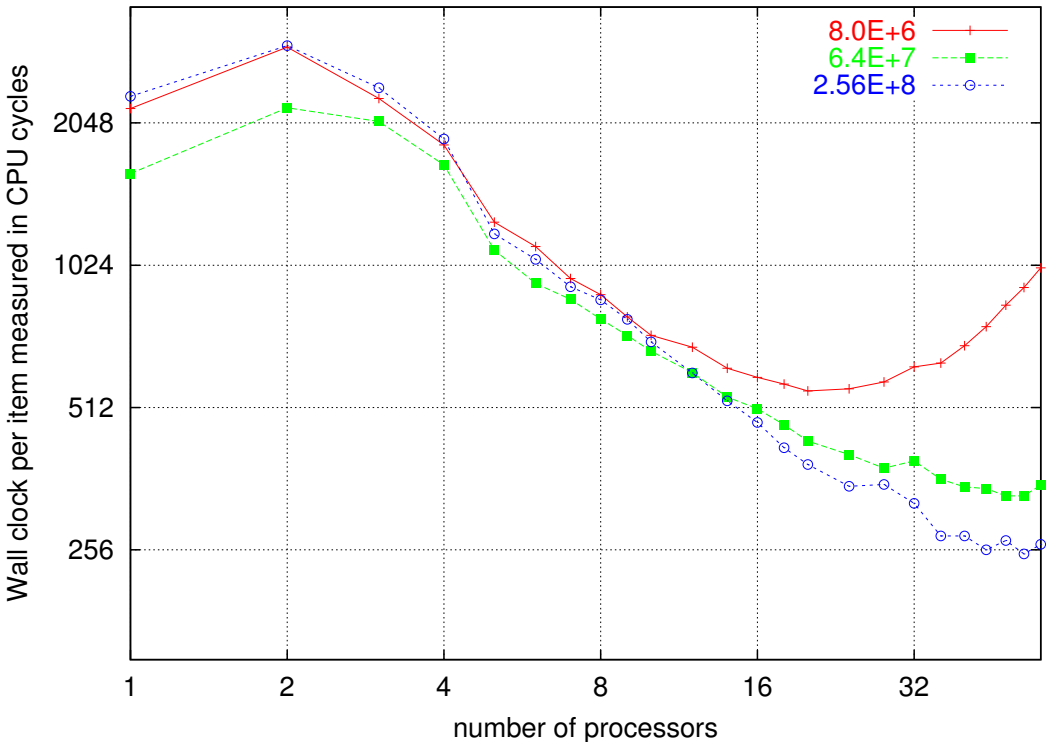
To see that this behavior also shows up in the experiments consider Figure 2. It represents the maximal input size that can be computed for both Sorting and List Ranking algorithms on the Icluster platform. Each node here has 256 MiB local memory and sequential version of List Ranking can only rank 5 million elements (resp. 10 million doubles for Sorting). For the parallel PRO algorithm, the more nodes we have, the greater input size we can compute and in particular we easily go beyond the input size that is tractable sequentially. In fact, by considering the global behavior shown in Figure 2 and disregarding the irregularity due to the discretization, we note that input size of PRO algorithms scales linearly in a wide range of processors. Figure 2 also indicates that Sorting scales a bit better than List Ranking. Indeed, if we focus on the descriptions of the algorithms 1 and 2, due to its recursive design and to the memory overhead of the independent set construction, the List Ranking algorithms requires more memory than Sorting.

So, the overall memory used by parallel version of both Sorting and List Ranking algorithms is in the same order of magnitude of the memory required by sequential versions. In PRO terms, these algorithms are then memory optimal.

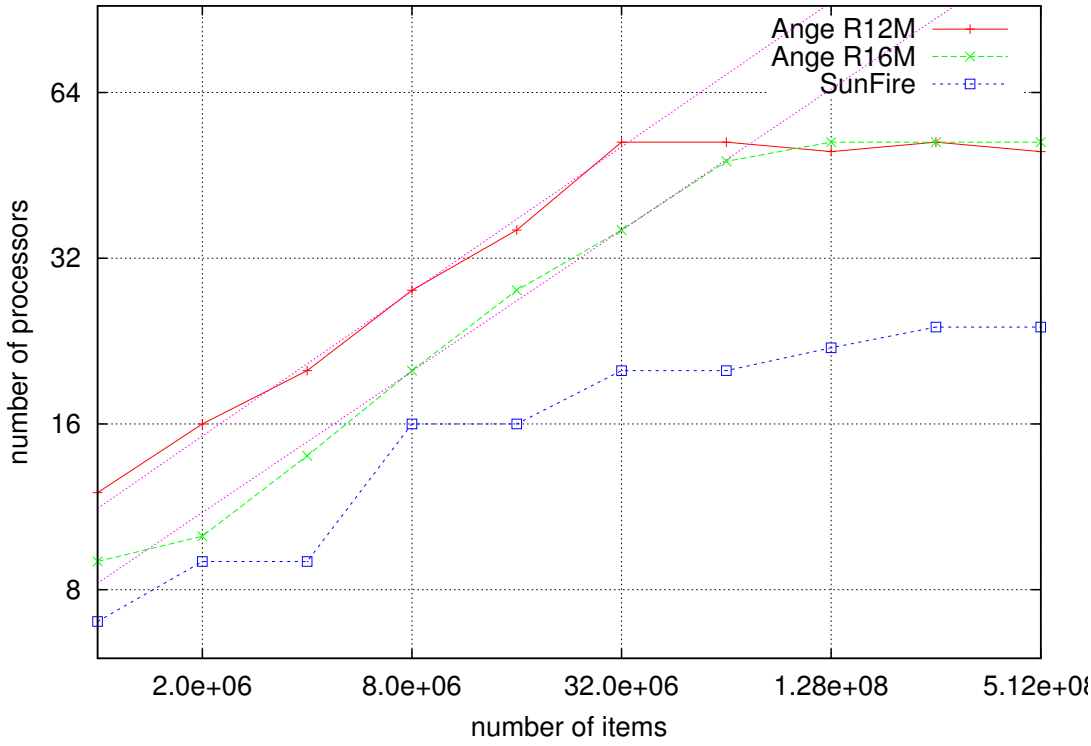
C. Granularity

To highlight the impact of the granularity on the efficiency of the PRO algorithms, we consider in Figure 3(a) the List Ranking results for three input sizes: 8, 64 and 256 million list elements. In this figure, we first distinguish the step that is due to the parallelization when going from 1 to 2 processors. Beyond 2 processors, there is an interval for which the number of cycles is independent on the input size and linearly decreases with $1/\mathrm{p}$. Since we obtain a regular and linear behavior and we achieve a very good speedup we can consider that the PRO requirements are fulfilled for these values.

Then, we notice that for each of the curves there is a value for the number of processors from which the time per item does not decrease neatly anymore and then even turns into a slowdown. This lack of efficiency is depending on the input size and mostly visible for the smaller inputs: the smaller the input size is, the early the slowdown effect is apparent. In Figure 3(b) we plot the number of list items against the maximum processor number that doesn't lead to a slowdown. This value can be considered as the limit for the granularity function as required by PRO.



(a) Input size 8, 64 and 256 million items on R16M



(b) Upper limit of processors for the on DSM machines

Fig. 3. Limits of scalability (List Ranking)

The figure illustrates the platform dependent relationship between the number of processors and the problem input size described as *Grain* function in the PRO model. As should be expected, the curves can be ranked according to the speed of the corresponding CPU. The higher the speed of the CPUs the less adding an additional CPU pays.

Before stopping at a *cut-off* of 52 processors (available were 56), the results for the Origin have an almost linear shape in the logarithmic scale. Thus for the range of *growth* in the processor number they correspond to a polynomial. We can approximate both R12M and R16M Origin results by a function of the form $p = \beta \cdot n^{\alpha}$. For the shown interpolations, we obtain $\alpha = 0.43$ and $\beta = 0.037$ for the R12M configuration and $\alpha = 0.42$ and $\beta = 0.034$ for the R16M respectively. These functions can be used as a PRO *Grain* functions for List Ranking on SGI SMP machines. Indeed, using such *Grain* functions, we can compute the optimal number of processors that can be used to treat a given number of elements by maintaining the same PRO algorithm quality. In the contrary, for a fixed number of processors, the inverse of the *Grain* value gives an evaluation of the input size that will efficiently use the available CPU resources. Both parts of the limit functions (growth and cut-off) are a little below the theoretical predictions: optimal would be an exponent for the growth of 0.5 and a cut-off at 56.

The interpretation of the curves for the SUN architecture is not as simple as for the Origin. This is due to the fact that there is not such a brutal cut-off below the optimum number of processors as for the Origin and that the transition between the growth and cut-off seems to be better tamed.

IV. Conclusion and future work

The PRO model provides a framework for the design and analysis of resource-optimal parallel algorithms. We have presented a set of experimental results that help validate the PRO model and demonstrate its practical utility. We showed that algorithms that satisfy all of the requirements of the PRO model can be efficiently implemented. For the two problems considered here, list ranking and sorting, the PRO algorithms are in fact CPU and memory optimal in comparison with the best sequential implementations. Our experimental study showed that the design of algorithms using the PRO model is largely independent of physical architecture and thus enables the prediction of algorithmic behavior on a variety of platforms.

In contrast to the architectures, the problems considered in this paper are *fine grained* in the sense that they act upon input that is composed of constant sized items. Work that focuses on fine grained algorithms in the more general area of cellular networks is currently in progress.

The PRO model assumes that the processors constituting a machine are *homogeneous* in terms of issues such as computing power, memory size, and available bandwidth. On the other hand, heterogeneous computational platforms, such as *grids*, are becoming increasingly important. Adapting the ideas in the PRO model to accommodate heterogeneous architectures is a direction we plan to explore in the future.

Acknowledgments

The authors would like to thank Assefaw Hadish Gebremedhin and the anonymous referees for their valuable remarks and suggestions concerning the contents and the form of this paper.

This work would not have been possible without the support and permission by the computing centers at the LERI in Reims, IMAG in Grenoble and LORIA in Nancy. Part of this research was financed by the PRST *Intelligence Logiciel* of the Lorraine Region and the *ACI ARGE* of the French Government.

References

E. Caceres, F. Dehne, A. Ferreira, P. Flocchini, I. Rieping, A. Roncato, N. Santoro, and S. W. Song. Efficient parallel graph algorithms for coarse grained multicomputers and BSP. In Pierpaolo Degano, Roberto Gorrieri, and Alberto Marchetti-Spaccamela, editors, *Automata, Languages and Programming*, volume 1256 of *Lecture Notes in Comp. Sci.*, pages 390–400. Springer-Verlag, 1997. Proceedings of the 24th International Colloquium ICALP'97.

Albert Chan and Frank K. H. A. Dehne. CGMgraph/CGMlib: Implementing and testing CGM graph algorithms on PC clusters. In Jack Dongarra, Domenico Laforenza, and Salvatore Orlando, editors, *PVM/MPI*, volume 2840 of *Lecture Notes in Computer Science*, pages 117–125. Springer, 2003. ISBN 3-540-20149-1.

Mohamed Essaïdi, Isabelle Guérin Lassous, and Jens Gustedt. SSCRAP: An environment for coarse grained algorithms. In *Fourteenth IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2002)*, pages 398–403, 2002.

Mohamed Essaïdi, Isabelle Guérin Lassous, and Jens Gustedt. SSCRAP: Soft synchronized computing in rounds for adequate parallelization. Rapport de recherche, INRIA, May 2004. URL `http://www.inria.fr/rrrt/rr-5184.html`.

Mohamed Essaïdi. *Echange de données pour le parallélisme à gros grain*. PhD thesis, Université Henri Poincaré, February 2004.

Assefaw Hadish Gebremedhin, Isabelle Guérin Lassous, Jens Gustedt, and Jan Arne Telle. PRO: a model for parallel resource-optimal computation. In *16th Annual International Symposium on High Performance Computing Systems and Applications*, pages 106–113. IEEE, The Institute of Electrical and Electronics Engineers, 2002.

Alexandros V. Gerbessiotis and Leslie G. Valiant. Direct Bulk-Synchronous Parallel Algorithms. *Journal of Parallel and Distributed Computing*, 22(2):251–267, 1994.

Isabelle Guérin Lassous and Jens Gustedt. Portable list ranking: an experimental study. *ACM Journal of Experimental Algorithmics*, 7(7), 2002. URL `http://www.jea.acm.org/2002/GuerinRanking/`.

Jop F. Sibeyn. Ultimate Parallel List Ranking. In *Proceedings of the 6th Conference on High Performance Computing*, pages 191–201, 1999.

L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.